Universitat Politècnica de Catalunya (UPC) – Barcelonatech

Facultat d'Informàtica de Barcelona (FIB)

Degree in Informatics Engineering
Information Technologies Specialization

OpenOverlayRouter with containers

Author:
Jose Ortiz Padilla
Director:
Albert Cabellos Aparicio
(Departament d'Arquitectura de Computadors)
Codirector:
Jordi Paillisé Vilanova
(Departament d'Arquitectura de Computadors)

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

January 23rd, 2018

Abstract

Open Overlay Router (OOR) is a tool to build Software defined networks based on Locator/ID Separation Protocol (LISP). There are some other tools to build overlay networks which use in Docker environments is widely extended. Among them, Open Virtual Switch (OVS) stands out. This report aims to describe the work performed in the scope of Docker with OOR. This work has two important parts. The first one is aimed to ease the development of OOR by introducing Docker tools and the second part is aimed to demonstrate how OOR can be used in Docker production environment like OVS does. OOR will add the LISP abilities to manage the containers from the networking perspective.

Index

Table of Figures

1. Background

In this section, we will briefly describe some technologies that are involved in this project to ease the reader to understand it.

1.1. Introduction

This document is a Report of the "OpenOverlayRouter with Containers" Degree Final Project developed in the "Facultat d'Informàtica de Barcelona (FIB)" which belongs to "Universitat Politècnica de Catalunya - BARCELONATECH" UPC. The project has been developed in the Computer Architecture Department of the UPC.

This project is based in two important technologies, on one hand, Docker. Nowadays when we talk about containers within the Computer Science society we mean Docker. Docker is an infrastructure technology, probably the most talked-about one only followed by Internet of Things, with a wide and fast adoption not only in development environments but also in production ones.

The impact can be shown in this figure:

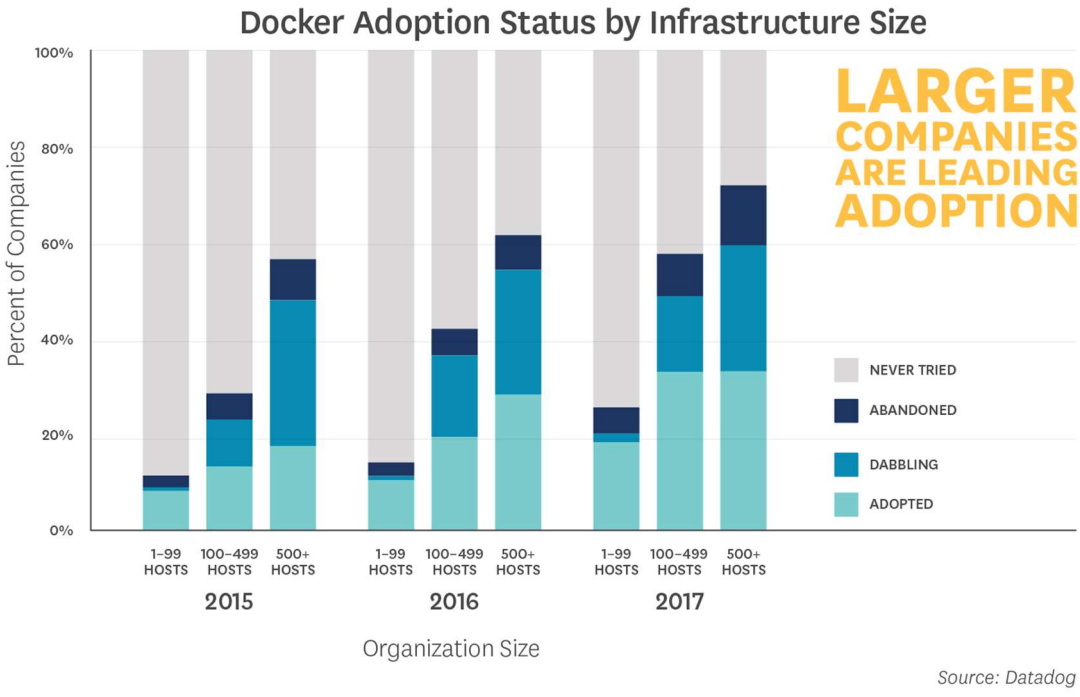

Figure 1 Docker Adoption Status

Source https://www.datadoghq.com/docker-adoption

One of the most important facts that have caused this deep adoption in the computer industry is the performance of Docker in comparison with the traditional virtual machines. Docker is light-weighted because it does not need a hypervisor, just the Docker daemon layer and the OS of the host are enough to run containerized applications.

The following figure describes the comparison between virtual machines (VM) and Containers in terms of performance (the longer is the vertical bar for an app, the worse is the performance):

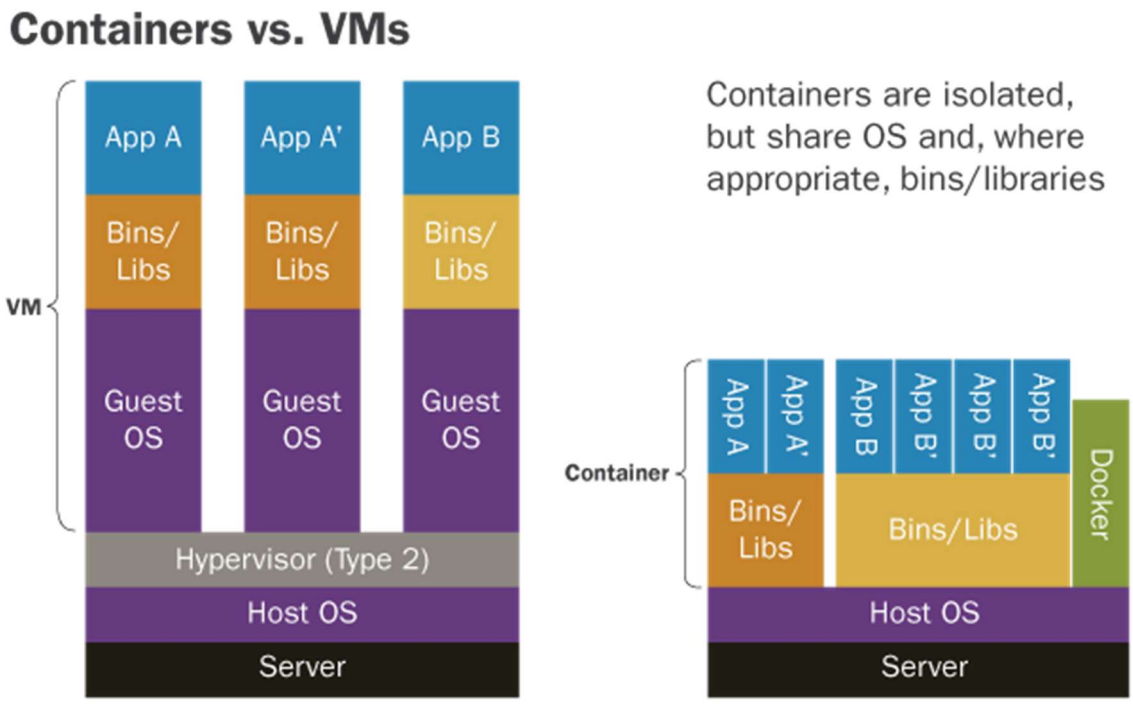

Figure 2 VMs Vs Containers

Source hub.docker.com

Other important features that have helped Docker in its fast adoption are the ease of migration of the containers, an easy deployment of applications thanks to this technology. In addition, Docker simplifies DevOps. DevOps is a new paradigm of development on which the developer has a part of the role that usually has been coupled to the operations team. The evolution of DevOps is as fast as the adoption of Docker. This paradigm allows developers to have Infrastructure as a code, an infrastructure that can be started just when it is required and destroyed by the development team itself. All these elements increase the productivity of the teams, and the productivity of the companies that use the combination of the mentioned technologies.

On the other hand, we have OOR. OOR stands for Open Overlay Router, which is an open-sourced router that implements the LISP protocol. LISP is also a wide-adopted protocol used frequently for VM mobility and for distributed datacenters. The features of LISP that help to the mobility of VM are described in the following diagram:

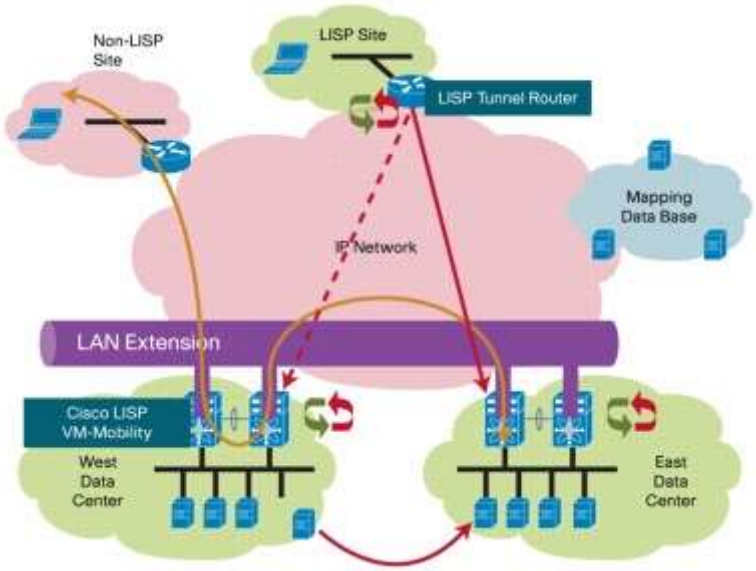

Figure 3 VM Mobility

Source www.cisco.com

In the figure above, we can see a LISP site which has Edge Networks on different regions and how easy is to migrate a server between those regions.
But OOR has a competitor widely extended in the Docker society called Open Virtual Switch (OVS) [1]. OVS, which is described in the state-of-art section below, is commonly used in Docker environments, also in Kubernetes (the most used Docker Orchestrator), some commercial products based on these technologies are also shipped with OVS. The following figure describes a common topology of Kubernetes that uses OVS:

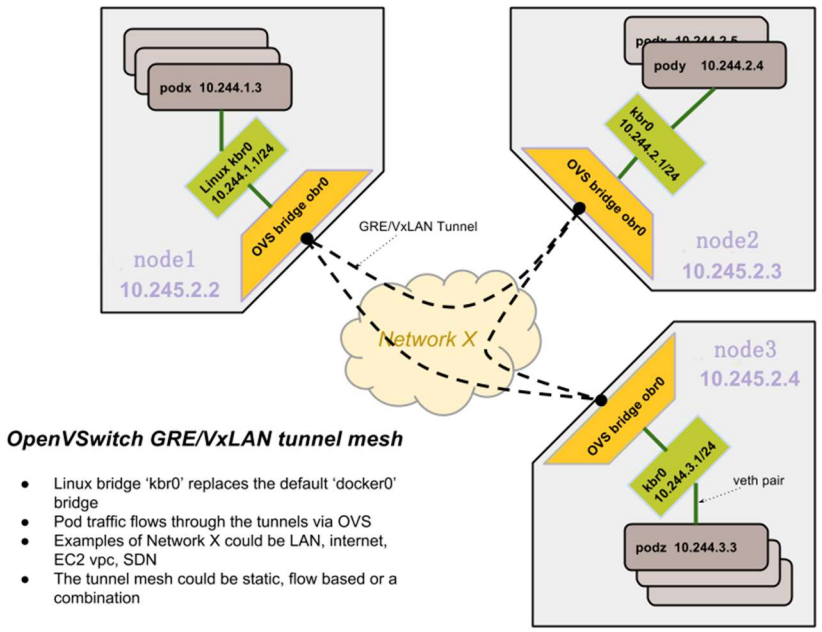

Figure 4 OVS and Kubernetes

[2]Source: kubernetes.io

Some people think that OOR has a better performance when used in conjunction with the Cisco's framework Vector Packet Processing (VPP) and an easier configuration than OVS. But there are just a few experiences of LISP in conjunction with Docker.
This project aims to create an environment of Kubernetes using OOR instead of OVS.

1.2. Docker concepts

Docker has two principal concepts, the containers and Docker itself.

1.2.1. Containers

Containers are a way to pack software in a format that can run isolated on a shared Operating System. These containers are not shipped with a full OS like VM. They only come with the libraries and dependencies that the application requires.

1.2.2. Docker

Docker is the leading software container platform. It is close to be a standard *de facto* to run and manage apps in isolated containers to get a better compute density and efficiency. It is commonly used by companies to build a complete pipeline to ship their applications from the development to the production environment in a faster, isolated and secure way as Docker explains on their whitepaper 'Docker for the Virtualization Admin' [3]

1.3. Software Defined Networking: SDN

A Software Defined Networking architecture defines how to allow separating the logic control to the physical devices. This architecture allows to centralize the control of the network. This architecture is easily explained defining the following three layers on which it is based:

SDN Applications

SDN Applications are programs that programmatically communicate, by using APIs, with the SDN Controller to build an abstracted network by using the controller's information.

SDN Controller

SDN Controller is an entity that receives requests and requirements from SDN Application layer and relies on the networking devices. It has an abstract view of the network and its statistics and events.

SDN Networking Devices

SDN Networking Devices control the forwarding and data processing capabilities for the network.

All of them can easily be figured out looking at this diagram:

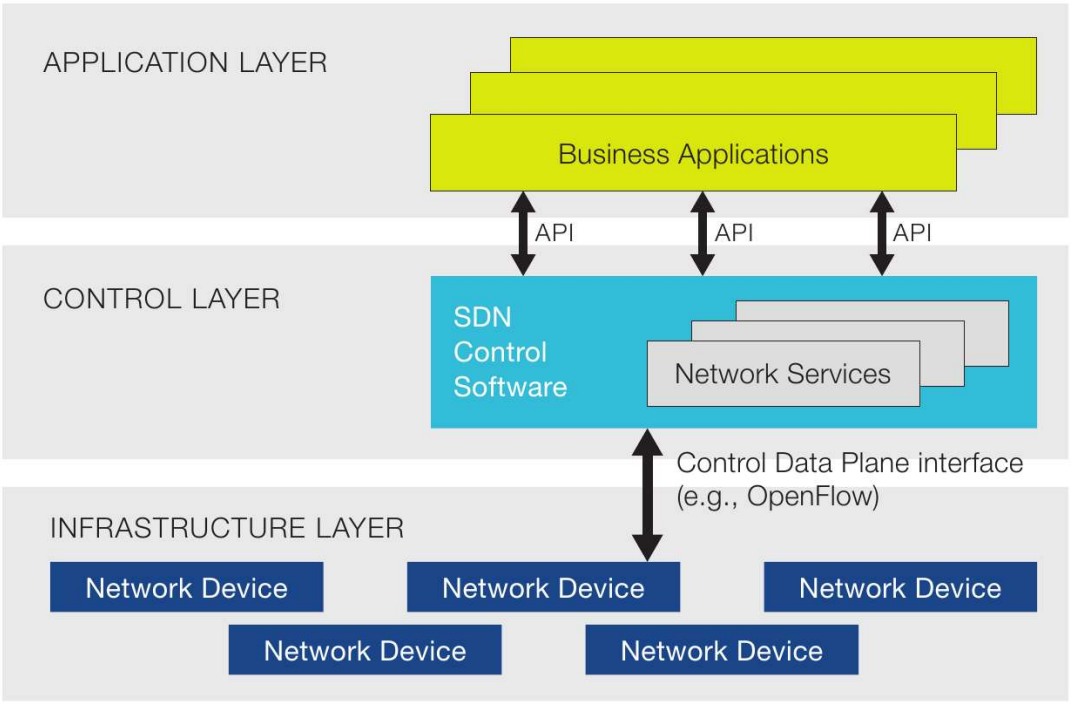

Figure 5 SDN Layers

Source: *Website: SdxCentral. Inside SDN architecture [4]*

1.4. Locator/ID Separation Protocol: LISP

LISP is a new semantic for IP addressing based on a network architecture and set of protocols. To say it briefly, LISP, as D. Meyer explains in the LISP tutorial [5], is:
Locator/ID Separation Protocol
Ground rules for LISP [2]:

- Network-based solution
- No changes to hosts whatsoever
- No new addressing changes to site devices
- Very few configuration file changes
- Imperative to be incrementally deployable
- Address family agnostic

Lisp is based on two planes:

- Data plane
    - Design for encapsulation and tunnel router placement
    - Design for locator reachability
    - Data-triggered mapping service
- Control plane
    - Design for a scalable mapping service

An usual topology that uses LISP can be described with the following figure:

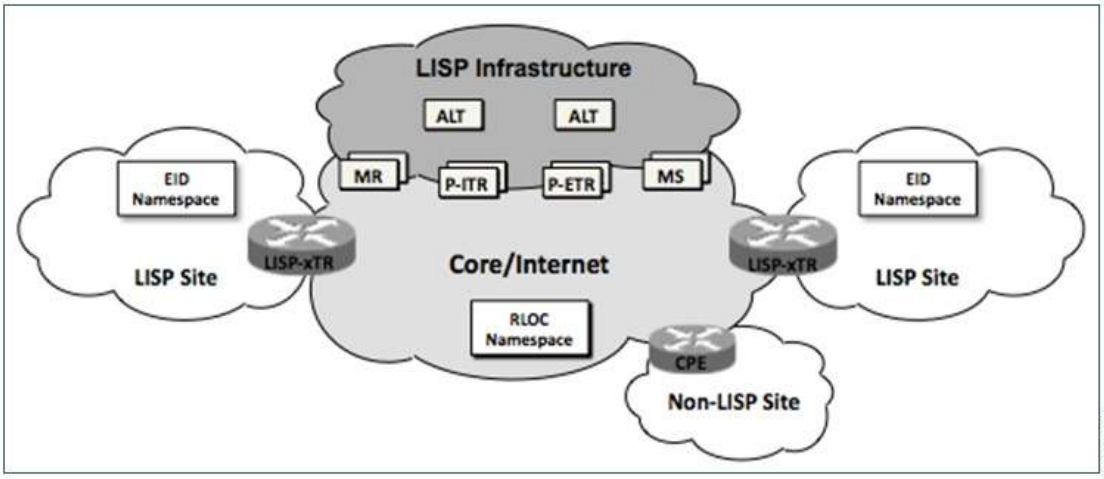

Figure 6 LISP Infrastructure

Source www.cisco.com

To ease the reading of this report, all the relevant topics from LISP (MS, MR, EID, xTR, RLOC) that are involved and used in this project are described in sections below. The main idea is that LISP has on one hand the locators, and on the oohter hand the edge networks. It does not matter where the LISP site is because it will be reached by other routers using the locator's (RLOC) namespace.

1.5. Open Overlay Router OOR

OOR [6] is an Open-source implementation to create programmable overlay networks using the LISP control-plane.
OOR leverages LISP to map overlay identifiers to underlay locators, and to dynamically tunnel overlay traffic through the underlay network. OOR is a portable, flexible, and extensible overlay solution that can be run in user space in multiple platforms (Linux, Android, and OpenWRT) and now also in Docker [8].

2. Problem formulation

There are a lot of SDN tools to build overlay networks for Docker. Several companies have developed their own tool. Furthermore, OVS is accepted for the community as a base product to develop SDN on the top of it.

But no company has developed a network overlay relying on the power of LISP to build SDNs within Docker.

For the reasons above, we want to design a robust Network overlay product based on OOR and LISP [7]. In addition to this, we want to make OOR more light-weight, usable and portable. This would increase the interest of the community and the industry on both products.

3. Context

The focus of this section is to explain the areas of interest of this project and how them will affect the stakeholders of the project.

3.1. Areas of interest

3.1.1. Containerize OOR

The first step of the project is to containerize OOR [7] using Docker. This is in the OOR roadmap since 2015 and now it will be developed.
It is the first step as it is desired to build all the other parts of the project based on Docker.

3.1.2. Design of an Overlay Network for Kubernetes and Docker with OOR

OOR and LISP are powerful SDN tools which have not been used yet with a container environment.
As Kubernetes is the most used orchestrator for Docker, this project aims to build a network overlay using OOR which leverage the LISP abilities to create a SDN on which the containers should reside.
There are some requirements that a network overlay must accomplish to have the ability of being used in Kubernetes:

- All containers should be able to communicate with each other over IP address.
- All nodes should be able to communicate with the containers that are running inside that node over IP address.
- All containers should be able to communicate with the Kubernetes daemons that are running in the master node.

3.1.3. Prototype of the overlay networking for Kubernetes and Docker

After designing how OOR will be deployed to build a SDN for Kubernetes, the aim of the project is to build a prototype to show how powerful OOR and LISP to create SDN for containers are.

3.2. Stakeholders

3.2.1. Audience

The audience of this project is people who are developing and investigating SDN, LISP and OOR. The goal is to create an industry-supported paradigm on how to enable software defined networking using LISP to orchestrate Docker containers. That is why the main target are researchers.

3.2.2. Users

The goal of this project is to build a network overlay for Kubernetes, then the potentially final users are all the Kubernetes community.

3.2.3. Beneficiaries

The LISP and OOR researchers, also all the OOR community, are the beneficiaries of this project. A whole lifecycle of continuous development/continuous integration will be delivered to them to ease the ability to test the new features before publishing a new version of OOR. A lot of developers have already downloaded parts of this project as it is shown in the following figure:

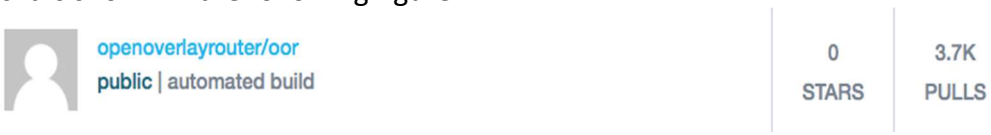

Figure 7 OOR Docker Container Statistics Since December 2017

Source DockerHub

4. State-of-the-art

In this section, we will discuss about some tools that are used as a SDN for containers. No one of these technologies uses LISP, but are now commonly used by the community and some enterprises to provide a SDN in a container environment. There are a lot of SDN under commercial licenses. We will just mention the products that are commonly used.
This project is the first attempt to containerize OpenOverlayRouter [6]. That is why there is no information related with building a SDN for Docker orchestrators based on LISP. The following are other alternatives to build SDN to ship and orchestrate Docker Containers.

4.1. Open Virtual Switch

Open Virtual Switch [8] (OVS) is an open-source project of a multilayer network switch. It is designed to enable network automation and to support standard management interfaces and several protocols. In addition to this, it is designed to support transparent distribution across multiple physical servers which allow to create cross-server switches. OVS is the default network switch for a lot of Hyper-visors (such as Xen, OpenNebula, Openstack…) and several Docker orchestrators (such as Kubernetes)
The architecture for OVS can be observed here:

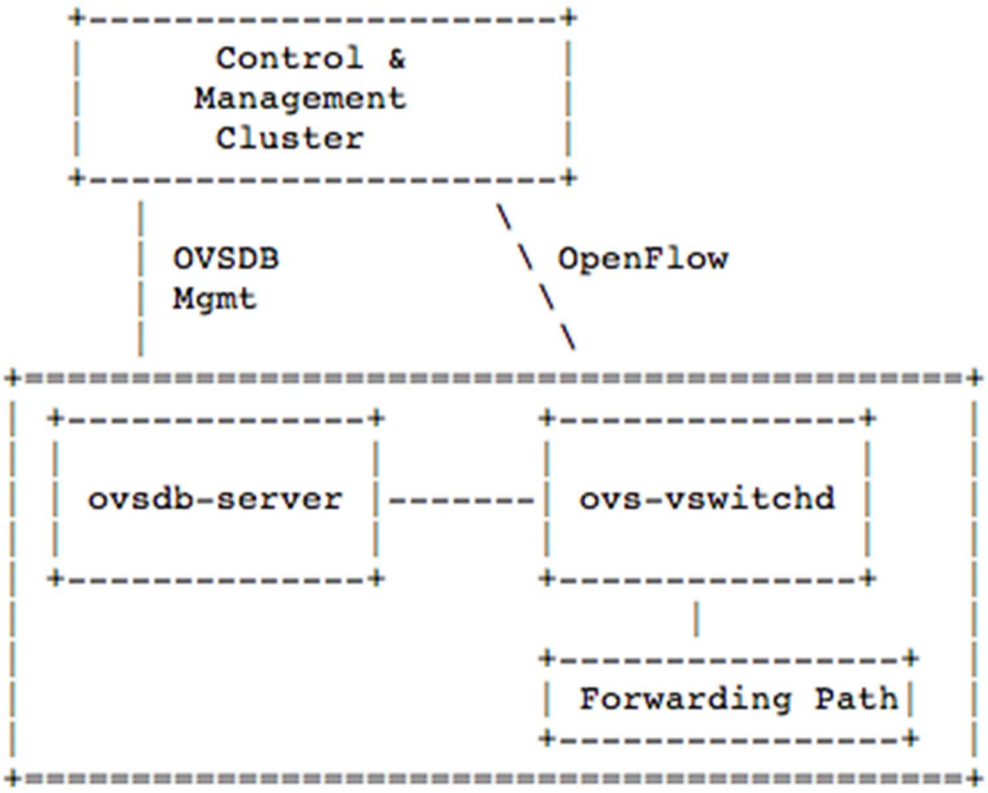

Figure 8 OVS Architecture

Source ovs at github.com

4.2. Nuage

Nuage, a product from Nuage Labs, is a SDN product to virtualize the network infrastructure of a data center and to connect the compute resources which have been created. It uses OVSDB (from OVS) as a management base to distribute the information of switching and routing to the hypervisors within the data center network.

The following figure is useful to describe how it works:

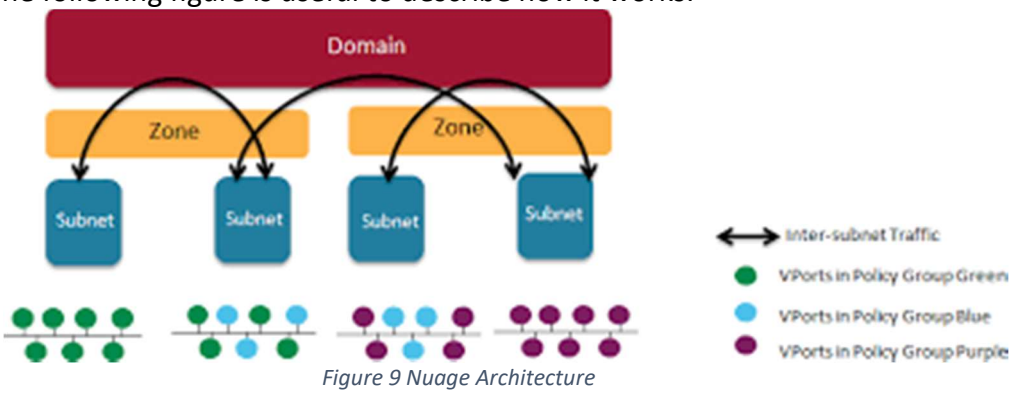

Figure 9 Nuage Architecture

Source: www.nuage.com

4.3. NSX

NSX is an VMWare project which aims to be a network virtualization tool for a data center based on software.
NSX allows to create complete network topologies using software and make them available to the hypervisor layer abstracting them from the underneath physical hardware. All the components of a network can be easily provisioned in minutes without modifying the application. It can be integrated with all VMware products and in addition with several Docker orchestrators.
The following figure is to describe the architecture of NSX working as Kubernetes Network driver:

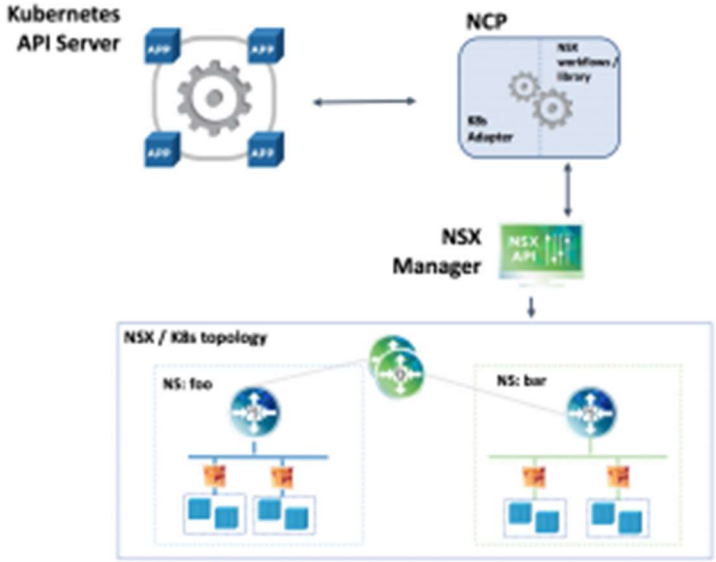

Figure 10 NSX Architecture with Kubernetes

Source www.vmware.com

5. Scope of the project

5.1. OOR in containers

As mentioned above in the context section, the scope of the project is to containerize OOR in Docker. In the scope, it is also included to build a complete CD/CI lifecycle for OOR [6]. This point will decrease the time for production of new features.

5.2. Analyze of how to use OOR as a Kubernetes Network Overlay and demo

Once the OOR will be within a Docker container, the next topic of the project is to build a network overlay driver to use OOR in Kubernetes. This topic may be difficult and tricky but the final goal is to build an environment to demonstrate the benefits of using OOR and LISP as network paradigm.

6. Methodology

Often the container projects are driven using the Agile development methodology. It is the best-fitted methodology but this project will not be developed by a team. This means that only a few tools will be used.

6.1. Test Driven Development

Test driven development (TDD) perfectly fits the requirements of the project. The idea is to build an initial prototype and to develop all the required features in every iteration of development.

6.2. Short development lifecycle

In order to reach the goal of the project, the development lifecycle should be short to allow researchers to test every new feature of the Docker container.

6.3. Intensive client feedback

The client is the researchers community. They are important to achieve all the goals of the project. In all the Agile methodologies, the client feedback is important. This case too. In fact, it may be more important because the researchers may develop some of it parts.

6.4. Development tools

Git and Github will be used to develop the project. All the sources will reside on Github and Git will be used to interact with it. For this project, the Dockerhub will be used as a container image repository
To implement the project, at least, four VMs will be required to build all the environment.
To write down all the documentation, Microsoft Word will be used. In addition to make available in an easy way the technical documentation for to the community, the wiki of the project in Github will be used.

In addition to this, a meeting with the directors of the project will be hold every two weeks. The aim of those meetings is to track the progress and overcome any deviation of the project if it is required.

6.5. Validation methods

A weekly meeting will be scheduled with the project tutor and a meeting with the researchers every two weeks.

7. Schedule

7.1. Estimated project duration

The project must not last more than six months. The project has started just after its inscription, which was in July, and would be finished before January 22th when the presentation period begins.

7.2. Considerations

Although the intention is not to delay any of the milestones, the initial planning could be revised if any of the phases lasts less than the expected. In addition to this, we have considered the fact that during August nearly all the components of the project were on holidays.

8. Project Planning

In this topic, all the stages or phases of the project will be described.

8.1. Project planning and feasibility

All the phases and milestones have been planned, although they might change. This phase was part of the GEP matter. This includes the following items:

- Project Scope & Context
- Project planning
- Project Budget
- State-of-the-art

The following resources will be used:

- Human resources:
  - The Product Manager which will be responsible for the roadmap and documentation of the Containerization of OOR and the Kubernetes Network driver.
- Hardware resources:
  - Laptop (Macbook Pro from early 2009)
- Software resources:
  - Microsoft Word to write down all the documentation
  - GanttProject to create a visual planning.

8.2. Project analysis and design

The main goal of this stage is to perform an analysis and design of OOR with Containers. We must define the requirements and the features that the final demo requires for working properly and, furthermore, define the use cases for it.

In this stage, the state-of-the art will be finished.

The following resources will be used:

- Human resources:
  - The Product Manager which will be responsible for the roadmap and documentation of the Containerization of OOR and the Kubernetes Network driver.
- Hardware resources:
  - Laptop (Macbook Pro from early 2009)
- Software resources:
  - Microsoft Word to write down all the documentation

8.3. Project iterations

Each iteration is divided into analysis, implementation, test and preparation for the next iteration.

8.3.1. Deep dive into LISP and OOR

In order to perform this deep dive, it is required to spend some time reading papers from LISP and the documentation of OOR [9]. It is expected to share doubts with the researchers.

The goal of this iteration is to acquire knowledge about LISP and OOR.

The following resources will be used:

- Human resources:
  - A Software engineer to gather all the information required to build a solution for the project.
- Hardware resources:
  - Laptop (Macbook Pro from early 2009)
- Software resources:
  - Github

8.3.2. OOR Containerization

It is expected to perform several sprints in this stage. In every sprint, an analysis of the situation will be done and an afterwards implementation of OOR within a Docker container and several tests to ensure that it fulfill all the requirements gathered during the analysis.

The goal of this iteration is to provide to the community a Dockerized version of OOR.

The following resources will be used:

- Human resources:
  - A Software engineer to build OOR within a Docker Container. It is required he/she may be involved with some Operating System concepts and with communication topics.
- Hardware resources:
  - Laptop (Macbook Pro from early 2009)
- Software resources:
  - Github
  - A plain text processor to write the specification of the OOR container

8.3.3. Deep dive into Kubernetes Network Drivers

The main task of this stage is to gather as much information as possible from how the Kubernetes Network Drivers work. It is expected to read some papers from Google and information shared by Google in Github about this topic.

The goal of this iteration, again, is to acquire a deep knowledge of Kubernetes and its Network drivers.

The following resources will be used:

- Human resources:
  - A Software engineer to gather all the information required to build a solution for the project
- Hardware resources:
  - Laptop (Macbook Pro from early 2009)

- Software resources:
    - Github

8.3.4. Build a demo about OOR as Kubernetes Network Driver

It is expected to do several sprints in this stage. In every sprint, an analysis of the situation will be done and afterwards demonstration about how OOR can work as a part of a Kubernetes Network Driver. Then, several tests to ensure that it fulfill all the requirements gathered during the analysis will be performed.

Every Sprint will imply:

- Analysis of the situation
- To perform an implementation of OOR as a part of a Network driver
- Test
- Preparation for the next sprint

The goal of the stage is to build a demo to show the driver designed in the previous stages.

The following resources will be used:

- Human resources:
    - A Software engineer to build a demonstration about how OOR can work as a part of a Kubernetes Network Driver. It is required he/she is involved with Docker, Kubernetes and some other Operating System concepts and with communication topics.
- Hardware resources:
    - Laptop (Macbook Pro from early 2009)
- Software resources:
    - Github
    - A plain text processor to write the specification of the OOR container

8.4. Final Stage

This stage includes two tasks:

- Creation of a final presentation
- Document and report to the community all the work done.

The final stage consists on the development of a report on how this Network

Driver for Kubernetes should be build. In addition, at this stage, the final presentation will be prepared.

9. Estimated Time

The following figure depicts the amount of time of dedication estimated for each stage of this project:

Stage	Dedication (hours)
1.- Project planning and feasibility	80
2.- Project analysis and design	90
3.1.- Deep dive into LISP and OOR	110
3.2.- OOR Containerization	80
3.3.- Deep dive into Kubernetes Network Drivers	180
3.4.- Build a demo about OOR as Kubernetes Network Driver	100
4.- Final Stage	100
Total	740

10. Gantt Chart

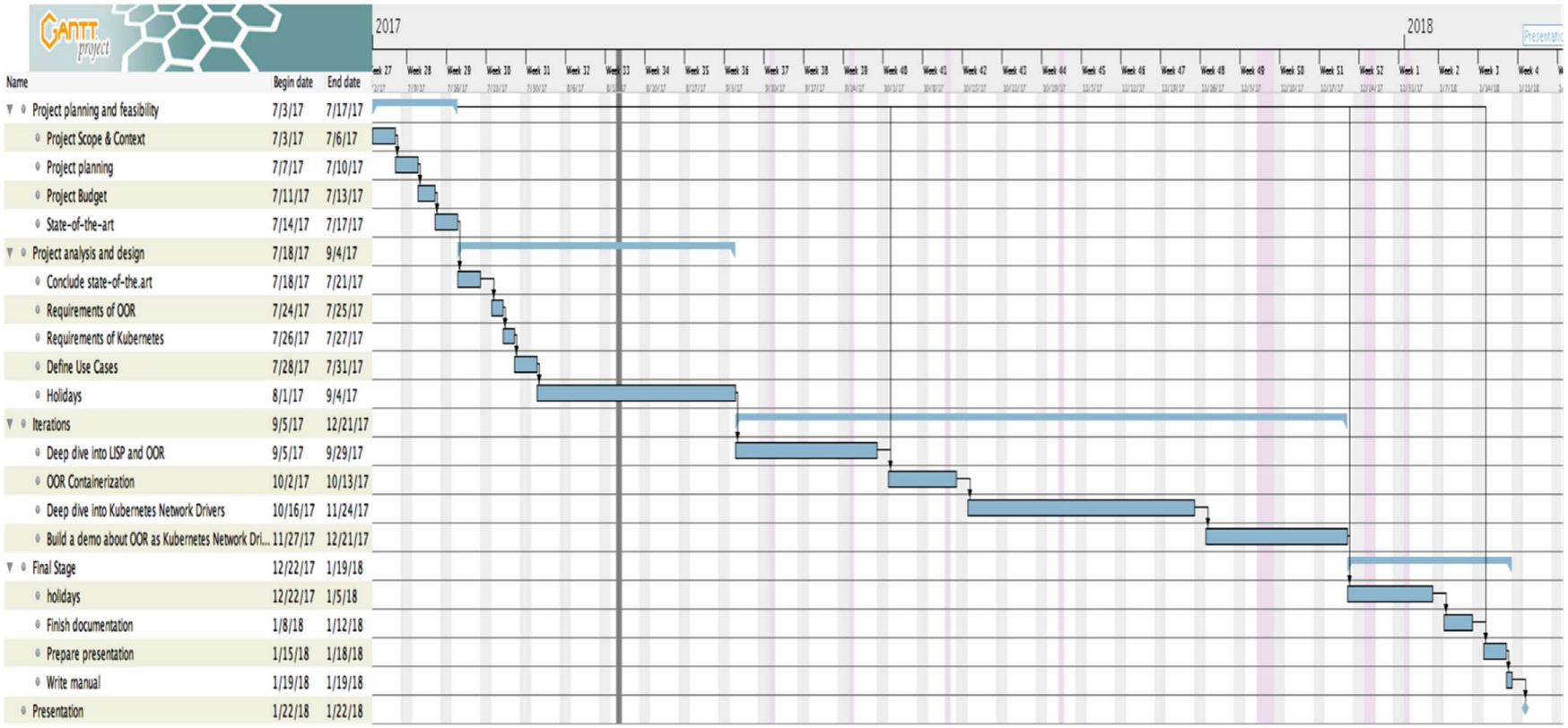

Figure 11 Gantt Chart

11. Possible obstacles and solutions

This project requires a deep knowledge of LISP and OOR. At the beginning of the project, the support of the researchers will be very important to ease the learning of OOR and LISP to the development team.

When the development team has the proper knowledge of the mentioned technologies, a prototype should be designed and implemented. This part of the project is the most difficult one and it will imply that the researchers test the prototype very oftenly.

As no one in the community has used OOR neither LISP to build a SDN for containers, all the efforts must be done by the development team, so the researchers may be required to support issues on OOR when working together with Docker.

Thus, the following tasks of the project will be tracked more accurately:

- OOR containerization
  - It is important to monitor this stage because OOR in Docker want to be the base of the future development of OOR.
  - The main risk is to not consider every use case for OOR in containers and build a container that no one is going to use.
- Demo of OOR acting as Kubernetes Network Driver
  - The risks of this stage are the possible deviation in the planning.

12. Alternatives and Action plan

As described in the previous sections, an Agile methodology will be used. The most important point of using this kind of methodology is the ability to adapt the duration of any stage.

At the end of every stage, a meeting with the Director or the Co-Director of the project and a simple report on the development done will be shared with the open- source community. If in this meeting any change in the initial schedule is detected, the alternatives will be to increase or decrease the duration of the following stage. The only milestone which cannot be moved is the day of the presentation. As TDD is being used, we can redefine the specification of a stage, or sprint, to be able to reach the goals within the dates of the milestones.

A deadline for the project has been marked. There is no warranty to have a full working software to be use as Docker Network Driver but a demonstration can be prepared to complete the project.

13. Resources

In order to develop this project, the following resources will be used:

Hardware:

1 laptop

4 VMware virtual machines

Software:

DockerHub

GitHub

GanttProject

Kubernetes

OOR

Docker

Microsoft Word

14. Cost Estimation

14.1. Direct costs

This section is divided following the activities described previously. To develop this project, it is required to have different roles. A product manager, a software engineer (with some networking abilities) and a Network engineer are required to develop the scope and context. The first two roles will be performed by an undergrad student, corresponding to a junior salary. But the Network engineer correspond to a Ph.D. student and then a senior salary is assigned.

In addition to the Human resources costs, some computering resources will be used. The laptop, bought 8 years ago, used to develop this project has been already amortized, thus it represents a 0€ costs. There is an option to renew the laptop, the cost of opportunity is calculated as:

Laptop_price – Laptop_amortization Which is 1350€-(1350€*740h/4*365*24h) resulting 1321,3€. As the cost of opportunity is high, the laptop won't be

renewed. Thus, the only cost related with this laptop is the power consumption which will be described in a section below. The cost of amortization of using Microsoft Word is 79€*Time_using_it/3*365*24h (As software is amortized in 3 years). The amortization results in 2,22€ Which will be added to the total of direct costs.

14.1.1. Project planning

In this stage of the project, the Product Manager is the only one required. Then the direct costs of this stage are 80h*30€/h.

	Item	Cost
	80x 1h Product Manager	30€
Total		2400€

14.1.2. Project analysis

This stage, like the previous one, requires only the Product Manager. The costs of this stage can easily be calculated in this way 90h*30€/h.

	Item	Cost
	90x 1h Product Manger	30€
Total		2700€

14.1.3. Deep dive into LISP and OOR

In this stage, the Software Engineer must get a deep knowledge on LISP and OOR. As there are 110h scheduled to develop this task, the cost will be 110h*30€/h.

	Item	Cost
	110x 1h Software engineer	30€
Total		3300€

14.1.4. OOR Containerization

To develop this stage, and based on the knowledge gathered on the previous one, 80h will be required by de Software Engineer to containerize OOR.

	Item	Cost
	80x 1h Software engineer	30€
Total		2400€

14.1.5. Deep dive into Kubernetes and Kubernetes Network Drivers

In this stage, again, the Software Engineer must get a deep knowledge of a matter, Kubernetes in this case. As there are 180h scheduled, the cost will be 180h*30€/h

	Item	Cost
	180x 1h Software engineer	30€
Total		5400€

14.1.6. Demo

This stage is the most complicated because there are more people involved than in the other stages. There are 100h scheduled, 70 of these 100h are required to develop the demo by the Software Engineer. The other 30h are scheduled for the Network Engineer.

In addition, the cost of the virtual machines must be considered. The cost for this project is then 3995,05*100h/4*365*24h resulting 11,40€.

	Item	Cost
	70x 1h Software engineer	30€
	30x 1n Network Engineer	50€
Total		3600€

14.1.7. Final Stage

In this final stage, again, only the Product Manager will be required to develop it. 100h has been scheduled, thus the cost of this stage will be 100h*30€/h.

	Item	Cost
	30x 1h Product Manager	30€
Total		3000€

14.1.8. Total

	Item	Cost
	Project planning	2400€
	Project analysis	2700€
	LISP & OOR Knowledge	3300€
	OOR in Docker	2400€
	Kubernetes Knowledge	5400€
	Demo	3300€
	Final Stage	3000€
	Virtual Machines	11,40€
	Microsoft Word	2,22€
Total		22513,62€

14.2. Indirect costs

Two main indirect costs have been considered: the power consumption (electricity) and the Internet connection. To be able to use internet, 43€ are paid. The amortization is based on the daily use of internet for this project, which is usually about 8 hours. The formula will look like (43*total_hours)/(30*8) = 132,6. Regarding the power consumption, the price of the kwh is, in average, 0,12€. The laptop is expected to be working on any of the stages of the project. It consumes 0,055 kwh, thus the cost of the power consumption will be: 0,12€/kwh*740h*0,055kw= 4,884€.

	Item	Cost
	Power consumption (Electricity)	4,884€
	Internet	132,6€
Total		137,484€

14.3. Contingency costs

The contingency costs have been calculated as the 15% of direct and indirect costs. That is why we can avoid possible delays providing more budget to enable more dedication if needed. Therefore, 3395€ of the total budget will be for this kind of contingency.

	Item	Cost
	Contingency	3395€
Total		3395€

14.4. Incidental costs

To avoid a delay on the Demo, the budget will be increased for this stage to cover

the need of working more than 30h during two weeks. Thus, the cost if this incident occurs is 900€.

	Item	Cost
	30x 1h Software engineer	30€
Total		900€

14.5. Total Costs

	Item	Cost
	Direct Costs & Indirect Costs	22651,104€
	Contingency	3395€
	Incidental	900€
Total w/o VAT		29946,104€
Total w VAT		36234,79€

14.6. Control management

The goal of this project is to build a demo to show how powerful is OOR creating SDN using LISP. The project can have deviations since all the products that we will use are open source and any of them might not be set up and ready for our project.

As we have seen in the estimation of costs, the main effect on this project is the time dedication. Thus, the most probable cause of increasing the budget will be the need of increase the time dedication of the Software Engineer. The other stages of the project are accurately scheduled and there is no need to dedicate more time as they should not have deviations.

It is important to control accurately the following tasks:

- OOR Containerization
- Build a demo about OOR as Kubernetes Network Driver

The comparison between the estimated hours of dedication and the actual hours spent in the project will be used as the control of costs. The hours will be compared because, on one hand the main costs is the hours of dedication and in the other hand the amortization of the virtual machines is also calculated based on the time that they are being used.

If we compare the amount of hours that has been spent we can easily translate it to costs deviation.

15. Sustainability

15.1. Economic

To develop this project, we have considered setting an extra budget to cover unexpected events. The cost of the project is low. The main part of the budget are the human resources thus, it is competitive. Once the project will be finished, one can replicate the work done in just minutes, which is one of the points of using Docker.

This project is a part of the OpenOverlayRouter (OOR) project, which is related with the LISP project and both are sustained by the community. There is still another good point of this project: it will help the researchers to save money of their investors. Using OOR in conjunction with Docker will reduce the hardware that must be used to develop it and will increase the useful life of this hardware.

15.2. Social

The developers tend to use tools like Docker because their ease of migration and promoting to production environments development very fast. This project aims to show the power of OOR [9] when creating Software Defined Networking to the Community. Many people have interest in developing SDN using Docker. In fact, it is part of the roadmap of the OOR project [7].

This project is intended to help the researchers; no sector of the community will be harmed. In addition, we must show that OOR can run using the latest technologies like Docker. It will just provide new and innovative options for SDN using Docker.

Another good point of the project is that the researchers will spent less time than before testing and the lifecycle of the updates of OOR will be shorter than nowadays.

15.3. Environmental

To develop this project, there is no need to use anything else than a laptop and several virtual machines. In the Project Development stage, just the laptop will be used. In the next stages, the laptop and about four Virtual Machines will be used. Thus, the only environmental impact will be the use of electricity. In the “Estimation of Costs” chapter we describe that this cost is not remarkable. It is not expected to use paper in any stage of the project. All the work will be

performed on electronic devices. Some existing hardware resources of the researchers will be reused to create the virtual machines required for the development of the project.

The scope of this project is to fulfil the Roadmap of the OOR. It is difficult to find an improvement in terms of the environment for this solution as it does not exist yet. Although there are studies that tell us why using Docker Containers are good for the environment, the Container Journal has described its benefits [9]. In this work, they tell that Docker can save energy in two ways. On one hand, Docker allows to run more applications in the same hardware than any other technology. On the other hand, they explain that using Docker can breathe new life to old hardware. This breath is given because Docker is light-weighted and one does not have to worry about hardware compatibility issues. Therefore, it is expected to have a small environmental footprint. In fact, it is expected to reduce the real footprint that OOR has nowadays.

15.4. Sustainability Matrix

As we have decided to reuse existing hardware and we have estimated the environmental impact very accurately, 9 is a good mark. As we have estimated the human and hardware resources accurately, 8 is a good mark. To wrap up, as I think that this project will be useful to develop my networking and Docker abilities, it will be very useful thus 9 is a good mark.

When the project development will end, the sustainability matrix during the exploitation is very similar. By using Docker, some benefits in the environment will be achieved. The dedication of the researchers will be lower than the dedication today. So we have benefits in the economic impact. After the project end, is not expected to have a deep social impact because the community will already know how useful it will be.

Regarding the risks of this project, during the development of this project no environmental risk has been found that can have an environmental footprint. Regarding economic risks, we have found that OOR as a Kubernetes driver require some development to achieve all the goals, thus it will have economic impact. To conclude with the risks, no social risk has been found. Then the sustainability matrix is:

	Project development	Exploitation	Risks
Environmental	9	10	0
Economic	8	9	-2
Social	9	8	0

16. Test environment and configuration

To ease reading of the report, the configurations, tests and results have been separated from each of the parts of the project. The first topic is intended to be a technical glossary. The following parts will be the description of the work performed in OOR as Docker container, and the work performed in OOR as a Kubernetes Networking Driver.

16.1. Concepts

To understand the tests described in sections below, it is important to describe the following concepts. These concepts are widely used in the project, thus before talking about their use will be described in this section:

- Dockerfile
  A Dockerfile is the specification of a Docker Container. It includes instructions to let Docker create a new image automatically. It is a text file that contains all the commands a user could call on the command line to assemble the image.

- Docker Image
  Once a Dockerfile is build, a Docker image is created. It has tags to identify it. Every Docker host has a local registry where locally built image is stored. There are several registries in the cloud and on-premises. To push the local images to any kind of registry is a common used best-practice.

- Docker Container
  A Docker container is a live Docker image. When a Docker image is running, it is volatile: any change made on it, we lost it when the container dies.

A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. Available for both Linux and Windows based apps, containerized software will always run the same, regardless of the environment. Containers isolate software from its surroundings. For example, differences between development and staging environments and help reduce conflicts between teams running different software on the same infrastructure [10].

- Docker-compose
  Docker-compose is a tool for defining and running applications that consist on one or more containers. It is a yaml file in which the services, like the network, can be configured. With this file and by running a simple command, the application and its services would start. [10]

- Kubernetes
  Kubernetes is an open-source system for automating deployment, scaling and management of containerized applications.

- Container Network Interface (CNI)
  CNI consists of a specification and libraries for writing plugins to configure network interfaces in Containers.

- RLOC
  Routing Locators is a concept of Open Overlay Router (OOR) to name hosts in transit networks.

- EID
  Endpoint Identifiers is a concept of OOR to name hosts in edge networks.

- MapServer/MapResolver (MS/MR)
  MS/MR is one of the available configurations of OOR. When OOR is configured as MS/MR it is used by other nodes to register their EID. When all the nodes have their EID registered, MS/MR is used to tell each node how to connect with the others.

- xTR
  xTR is another available configuration of OOR. It mainly has two important parts: the RLOC where we can find the MS/MR and the part of the configuration where the EID are declared.

- MN
  MN is another configuration of OOR. The main difference with xTR is that the EID is associated with the interface lispTun0.

- Ansible

  Ansible is a tool for automation. It allows to remotely perform actions in the servers defined in a inventory.

  It has two ways of working, by using the command line or by running ansible playbooks. Ansible playbooks are yaml files that contain one or more actions and the scope of the servers where it will be run.

16.2. OOR as Docker container

16.2.1. Introduction

As briefly mentioned in previous sections, we have used a laptop and four virtual machines.

This environment is described in the following diagram:

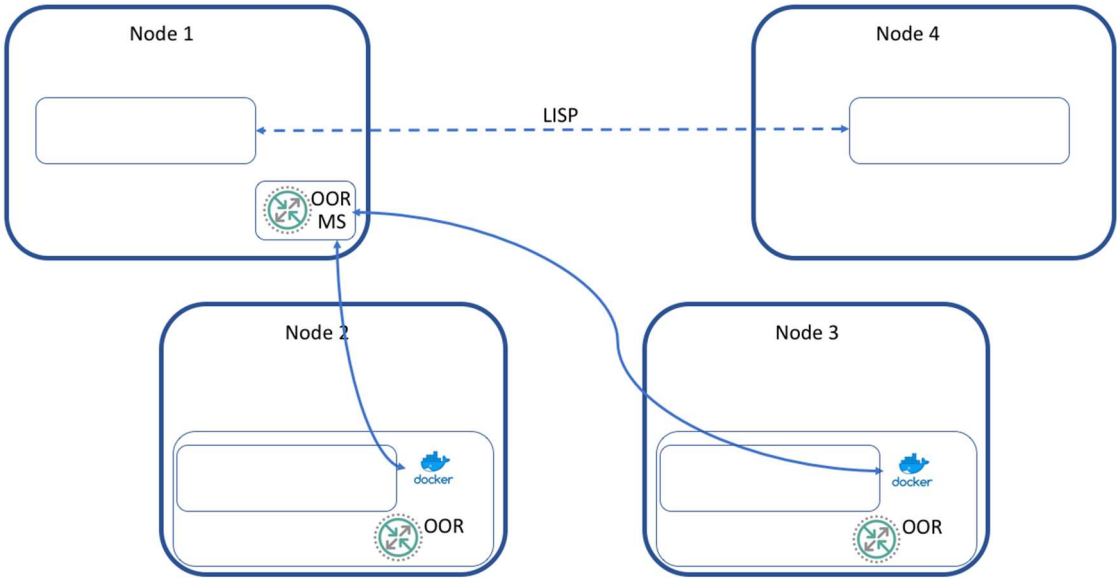

Figure 12 Schema of the test environment

Own compilation

In this environment, one virtual machine is used as OOR MapServer/MapResolver.

In the configuration that will be shown in the section below, we can see that we are defining the network in which the OOR xTR can publish their EID prefixes.

The other parameters of the configuration are used to register each EID that belong to an xTR.

The configuration of all the OOR xTR is shown in the following section.

The difference between each xTR configuration is the EID prefix that is different for each server. The nodes, marked in the figure above as node 2 and node 3, will act as LISP routers (OOR). These nodes will publish different EIDs. One interface of the node 1 will use node 2 as gateway to reach the LISP network, and node 4 will use node 3 as gateway to reach the other EID.

With the configuration shown in the following section, the nodes node 1 and node 4 will have a connection using LISP through the OOR working in Containers of node 3 and node 4.

16.2.2. Topology of these tests

This test aim to check that OOR can work as xTR wthin a Docker Conteiner and a complete lifecycle can be used. Since an OOR developer pushes its code to Github until watchtower discover the new version and stops the working container, it pulls the new image and starts a new container with the latest version of OOR.

Node 1 will have the configuration for MS/MR which has been detailed above. Node 2 and node 3 will run the OOR Docker containers and watchtower to finish the lifecycle from the development to the deployed solution.

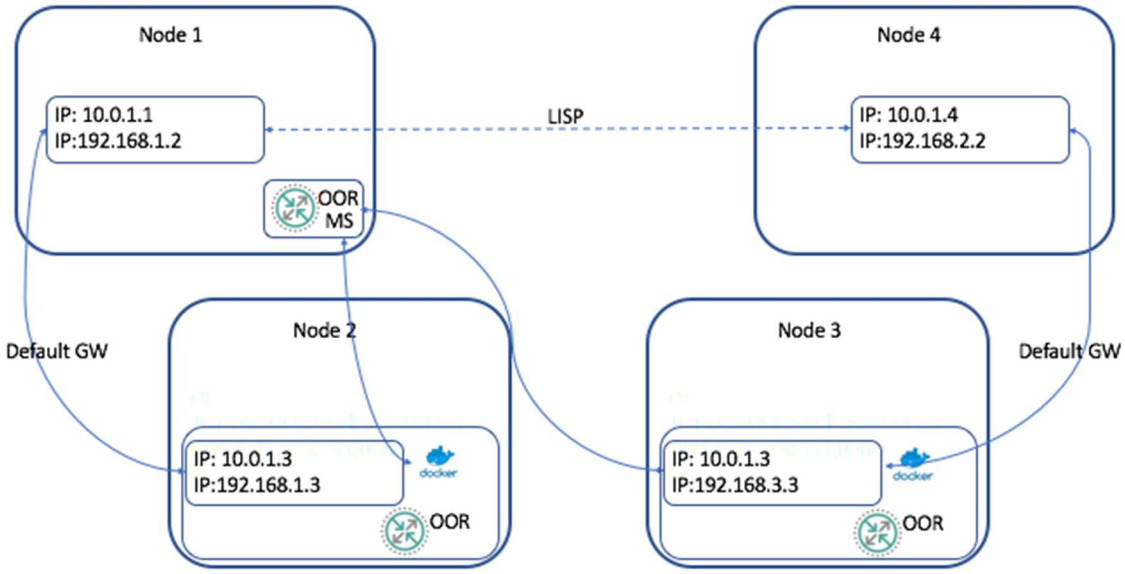

Figure 13 OOR in containers Tests

Own compiltion

16.2.3. Testing Dockerfile and Docker Image

The base concept of the test will be described in this section.

16.2.3.1. Dockerfile

The specification for the OOR image consists of two important parts, the preparation of the container to run OOR and the execution of OOR itself. The execution of OOR is performed in the start.sh script. The CMD clause in the Dockerfile determines which command will be issued inside the container. The ENV variables are also important to mention because they allow to change important parameters of OOR in every execution of a container.
In Docker, each line of the Dockerfile becomes a layer in the image you want to build. So, the more lines are included in the Dockerfile, the more space will require the image to work.

```
FROM ubuntu:16.04
MAINTAINER jose.orpa@gmail.com
ENV TERM=xterm
RUN apt-get update -y && apt-get install software-properties-
common -y && apt-add-repository ppa:ansible/ansible -y && apt-get
update -y && apt-get install -y dialog apt-utils && apt-get
install -y build-essential git-core libconfuse-dev gengetopt
libcap2-bin libzmq3-dev libxml2-dev ansible && apt-get autoclean
&& apt-get autoremove
RUN echo 'net.ipv4.conf.default.rp_filter=0' >> /etc/sysctl.conf
RUN echo 'net.ipv4.conf.all.rp_filter=0' >> /etc/sysctl.conf
RUN echo 'net.ipv4.ip_forward=1' >> /etc/sysctl.conf
RUN echo 'net.ipv6.conf.all.forwarding=1' >> /etc/sysctl.conf
RUN git clone git://github.com/OpenOverlayRouter/oor.git
WORKDIR /oor
RUN make
RUN make install
RUN rm -rf /oor
COPY files/*.conf /etc/
COPY files/*.sh /tmp/
COPY files/*.yaml /tmp/
ENV DEBUG <int 0..3>
ENV OPMODE xTR
ENV IPMAPRESOLVER <IP of the MapResolver>
ENV IPMAPSERVER <IP of the MapServer>
ENV KEYMAPSERVER <String>
ENV IPPROXYETRV4 <IP of the Proxy ETR IPv4>
ENV IPPROXYETRV6 <IP of the Proxy ETR IPv6>
ENV IPV4EIDPREFFIX <EID IPv4 Preffix>
ENV IPV6EIDPREFFIX <EID IPv6 Preffix>
CMD /tmp/start.sh $DEBUG $OPMODE $IPMAPRESOLVER $IPMAPSERVER
$KEYMAPSERVER $IPPROXYETRV4 $IPPROXYETRV6 $IPV4EIDPREFFIX
$IPV6EIDPREFFIX
```

Another important topic of the image is that the source code of OOR is downloaded each time you create a new version. In this way, the image will have always the latest stable version of OOR.
All the environment variables are used as parameters of the start.sh script, which it will be shown below, to modify the configuration of OOR before starting it.

16.2.3.2. Coding start.sh

The main goal of start.sh is to modify oor.conf using the environment variables, and run oor.
To configure oor.conf, Ansible has been chosen as automatic configuration tool. Before running oor, an ansible playbook is executed to set all the parameters in the configuration file.
The code for start.sh is:

```
#!/bin/bash
cat <<EOT >> /etc/ansible/hosts
[targets]
localhost              ansible_connection=local
EOT
sed -i s/DEBUG/$1/g /tmp/oor.yaml
sed -i s/OPMODE/$2/g /tmp/oor.yaml
sed -i s/IPMAPRESOLVER/$3/g /tmp/oor.yaml
sed -i s/IPMAPSERVER/$4/g /tmp/oor.yaml
sed -i s/KEYMAPSERVER/$5/g /tmp/oor.yaml
sed -i s/IPPROXYETRV4/$6/g /tmp/oor.yaml
sed -i s/IPPROXYETRV6/$7/g /tmp/oor.yaml
sed -i s/IPV4EIDPREFFIX/$8/g /tmp/oor.yaml
sed -i s/IPV6EIDPREFFIX/$9/g /tmp/oor.yaml
ansible-playbook /tmp/oor.yaml
ansible all -m lineinfile -a "dest=/etc/oor.conf state=absent regexp='^<'"
oor -f /etc/oor.conf
```

The ansible-playbook, is the tool used to modify the configuration file of OOR within the container during its startup. The playbook is based on a new feature of ansible that allows to modify blocks of text inside a file. The play-book looks like the following:

```
 -name: OOR Map Resolver
  blockinfile:
    dest: /etc/oor.conf
    block: |
      map-resolver       = {
            {{ var3 }}
      }
    marker: "<!-- {mark} map-resolver -->"
```

```
    backup: no
- name: OOR Map Server
  blockinfile:
    dest: /etc/oor.conf
    block: |
      map-server {
            address       = {{ var4 }}
            key-type      = 1
            key           = {{ var5 }}
            proxy-reply    = off
      }
    marker: "<!-- {mark} map-server -->"
    backup: no
- name: OOR Proxy ETR IPV4
  blockinfile:
    dest: /etc/oor.conf
    block: |
      proxy-etr-ipv4 {
            address     = {{ var6 }}
            priority    = 1
            weight      = 100
      }
    marker: "<!-- {mark} proxy-etr-ipv4 -->"
    backup: no
- name: OOR Proxy ETR IPV6
  blockinfile:
    dest: /etc/oor.conf
    block: |
      proxy-etr-ipv6 {
            address     = {{ var7 }}
            priority    = 1
            weight      = 100
      }
    marker: "<!-- {mark} proxy-etr-ipv6 -->"
    backup: no
- name: OOR Database Mapping IPV4
  blockinfile:
        eid-prefix         = {{ var8 }}
        iid              = 0
        rloc-iface{
           interface      = eth0
           ip_version      = 4
           priority       = 1
           weight         = 100
        }
      }
    marker: "<!-- {mark} database-mappingv4 -->"
```

```
    backup: no
 - name: OOR Database Mapping IPV6
   blockinfile:
    dest: /etc/oor.conf
    block: |
     database-mapping {
        eid-prefix        = {{ var9 }}
        iid               = 0
        rloc-iface{
           interface      = eth0
           ip_version     = 4
           priority       = 1
           weight         = 100
        }
     }
    marker: "<!-- {mark} database-mappingv6 -->"
    backup: no
```

16.2.3.3. How to get the image

To get the image, it is just required to run the following command:

```
docker pull openoverlayrouter/oor
```

It will make available the image in the local docker registry on the server where the command has been run.

16.2.4. OOR Docker Image usage

16.2.4.1. Setting up the environment

You can avoid this step using docker-compose files that can automatically create the Docker Networks for you. This feature is shown below in the docker-compose section.
To run the Docker Container for Linux operating directly, create first the Docker Networks:

One for RLOC:

```
docker network  create  -d macvlan --subnet=<RLOC Subnet> --
gateway=<Gateway> -o parent=<Interface> -o macvlan_mode=bridge
<network_name>
```

Another one for EID:

```
docker network  create  -d macvlan --subnet=<EID Preffix> -o
parent=<Interface> -o macvlan_mode=bridge <network_name>
```

And run the docker container:

```
docker create --net=<RLOC_Docker_Network_Name> --ip=<IP_RLOC> --
name <docker_name> -it --device=/dev/net/tun --cap-add=NET_ADMIN
--cap-add=NET_RAW -e IPV4EIDPREFFIX="<network>\/<mask>" -e
IPV6EIDPREFFIX="<network>\/<mask>" -e DEBUG="<int 0..3>" -e
OPMODE="xTR" -e IPMAPRESOLVER=<IP of the MapResolver> -e
IPMAPSERVER=<IP of the MapServer> -e KEYMAPSERVER=<String> -e
IPPROXYETRV4=<IP of the Proxy ETR IPv4> -e IPPROXYETRV6=<IP of
the Proxy ETR IPv6> -e IPV4EIDPREFFIX=<EID IPv4 Preffix> -e
IPV6EIDPREFFIX=<EID IPv6 Preffix> openoverlayrouter/oor:latest

docker network connect <EID_Docker_Network_Name> --
ip=<EID_IP_forContainer> <docker_name>

docker start <docker_name>
```

16.2.5. OOR Docker-Compose

16.2.5.1. Docker Compose Code

Docker-Compose is an easy and quick way to deploy containers. It has been developed by Docker and can e integrated with the Docker orchestrator which is called Docker Swarm.

We use Docker-compose because is a way to define all the resources required in a unique file. In addition, to run containers which are specified in a docker-compose file is easier than running it over the Docker daemon stand-alone.

The docker compose binaries let the user to create, destroy, start, stop and view the logs of one or more containers using just one command.

The following is the specification of the OOR container and the networks that it requires working.

```
version: "3"
services:
  oor:
   image: openoverlayrouter/oor
   cap_add:
     - NET_ADMIN
     - NET_RAW
   devices:
     - "/dev/net/tun:/dev/net/tun"
   networks:
     0rloc:
         ipv4_address: <ip_for_the_container>
     1eids:
         ipv4_address: <ip_for_the_container>
   environment:
     - IPV4EIDPREFFIX="<network>\/<mask>"
     - IPV6EIDPREFFIX="<network>\/<mask>"
     - DEBUG="<int 0..3>"
     - OPMODE="xTR"
     - IPMAPRESOLVER=<IP of the MapResolver>
     - IPMAPSERVER=<IP of the MapServer>
```

```
      - KEYMAPSERVER=<String>
      - IPPROXYETRV4=<IP of the Proxy ETR IPv4>
      - IPPROXYETRV6=<IP of the Proxy ETR IPv6>
      - IPV4EIDPREFFIX=<EID IPv4 Preffix>
      - IPV6EIDPREFFIX=<EID IPv6 Preffix>
networks:
  1eids:
    driver: macvlan
    ipam:
      driver: default
      config:
      - subnet: <EID IPv4 Preffix>
      driver_opts:
          parent: <linux host interface>
      macvlan_mode: bridge
  0rloc:
    driver: macvlan
    ipam:
      driver: default
      config:
      - subnet: <RLOC Subnet>
   driver_opts:
      parent: <linux host interface>
      macvlan_mode: bridge
```

16.2.5.2. Docker compose usage

```
docker-compose -f docker-compose-network.yml up -d
```

16.2.6. Building a Continuous Deployment/ Continuous Integration Environment

16.2.6.1. Defining the lifecycle

The lifecycle of a Continuous Deployment/Continuous Integration environment is started when a Developer modifies any of the code of an application and finishes after the deployment and integration of it, when the application is ready for production again.
In the use case of OOR, the researchers are responsible of modifying the source code of OOR.
The lifecycle will start automatically after they push the changes on their repository. It will build a new Docker image of OOR and it will also push this image to the Dockerhub, where the image will be available to everyone who want to test OOR easily.
Another process will watch if a new version of OOR is built, and when the new version is available, it will download the new image, stop the running container and start a new one with the same parameters.

16.2.6.2. Tools

The following tools are used to achieved the mentioned result:

- Editor
  Any Editor works in this solution.
- Github
  Github is a well-known Version Control Repository.
- Dockerhub
  Dockerhub is one of the well-known registries of Docker Images. It is easy to link a namespace in Dockerhub with a repository of Github.
- WatchTower
  Watchtower is a Docker container that can run in any hosts where the developers want to update OOR automatically.

16.2.7. Results of OOR Container

The tests of OOR in a Docker container were successful. We got an environment with a fully working LISP network from the Node 1 to the Node 4.

The nodes Node 2 and Node 3 were running OOR Docker Container, using the Docker-compose configurations to ease the tests, and watchtower to pull the latest version of OOR.

Using these steps, the test of upgrading OOR without any human action in the test environment was achieved. It is just required that a developer pushes a new code to Github to trigger all the workflow.

The workflow follows with the automatic build of the OOR image and the automatic push to the DockerHub. When the new version has been pushed to Dockerhub and it is available, the Watchtower tool watches it and it starts to pull the image to the servers. When the image has been downloaded, the Watchtower tool stops the running OOR container and starts a new one with exactly the same configuration. As the configuration of the OOR that runs within the container is based on the parameters which the container has been running, the configuration of the new container using the new image will be the same, thus the xTR node will register the same EID.

During the OOR Container restart process, the network connection will be lost between the nodes through LISP. But this process is very fast and it does not take more than 10 seconds to be finished.

16.3. OOR as Kubernetes Network Driver

16.3.1. Introduction

The configuration required to use OOR as a Kubernetes Networking driver has

two important parts: the part related with OOR and the Kubernetes/Docker part.

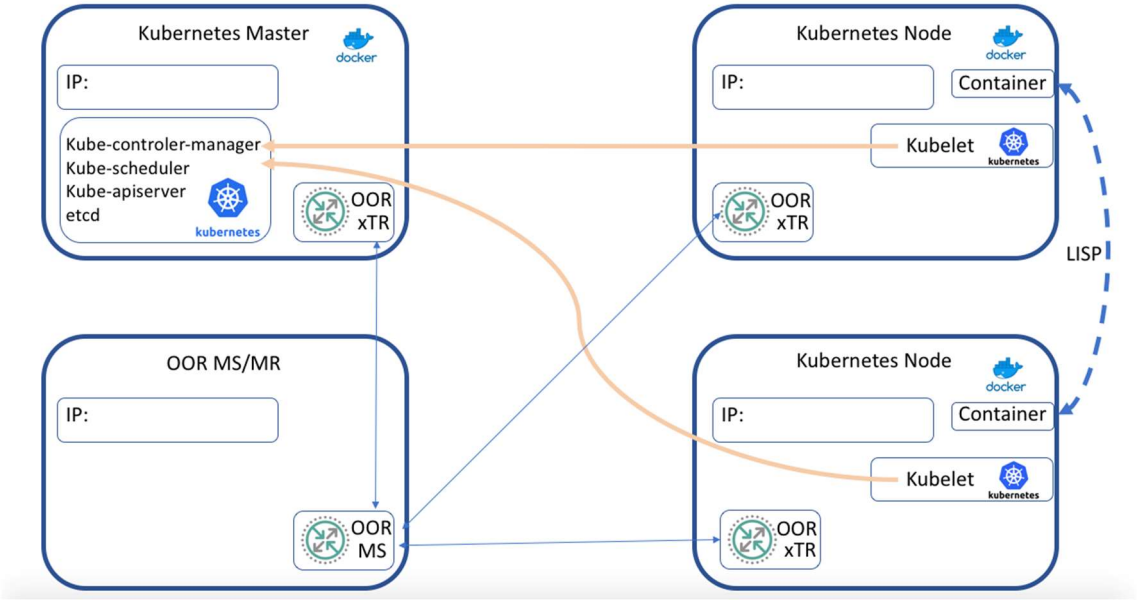

Figure 14 Kubernetes and OOR

Source own compilation

To start with the OOR part of the configuration, which corresponds with the OOR MS in the figure above, that is required is the configuration of the MS/MR which is the following:

```
##############################################
#
# General configuration
#
# debug: Debug levels [0..3]
# map-request-retries: Additional Map-Requests to send per map cache
miss
# log-file: Specifies log file used in daemon mode. If it is not
specified,
#   messages are written in syslog file
debug                   = 3
map-request-retries     = 2
log-file                = /var/log/oor.log
# Define the type of LISP device LISPmob will operate as
#
# operating-mode can be any of:
# xTR, RTR, MN, MS
#
operating-mode          = MS
# For the rest of this file you can delete the sections that does not
apply to
# the LISP device selected in operating-mode
##############################################
#
# MS configuration
#
# Control messages are received and generated through this interface
# Only one interface is supported
control-iface = ens224
# Define an allowed lisp-site to be registered into the Map Server.
Several
# lisp-site can be defined.
#
#   eid-prefix: Accepted EID prefix (IPvX/mask)
#   key-type: Only 1 supported (HMAC-SHA-1-96)
#   key: Password to authenticate the received Map-Registers
#   iid: Instance ID associated with the lisp site [0-16777215]
#   accept-more-specifics [true/false]: Accept more specific prefixes
#     with same authentication information
lisp-site {
    eid-prefix            = 10.244.0.0/16
    key-type              = 1
    key                   = lispmob
    iid                   = 0
    accept-more-specifics = true
}
lisp-site {
    eid-prefix            = 192.168.8.0/24
    key-type              = 1
    key                   = lispmob
    iid                   = 1
    accept-more-specifics = true
}
```

As is shown in the figure above, there are two LISP sites, one for the containers and another to allow communication through LISP between the nodes too.

Once we have the MS/MR working, the Edge networks should be configured in

every server. In the diagram above, this node are marked as xTR. The first block is the same for all xTR servers. It is shown in the following figure:

```
##############################################
#
# General configuration
#
# debug: Debug levels [0..3]
# map-request-retries: Additional Map-Requests to send per map cache
miss
# log-file: Specifies log file used in daemon mode. If it is not
specified,
#   messages are written in syslog file
debug                   = "3"
map-request-retries     = 2
log-file                = /dev/stdout
# Define the type of LISP device LISPmob will operate as
#
# operating-mode can be any of:
# xTR, RTR, MN, MS
#
operating-mode          = xTR
##############################################
#
# Tunnel Router general configuration
# Common for xTR, RTR & MN
#
# encapsulation: Encapsulation that will use OOR in the data plane.
Could be
#   LISP or VXLAN-GPE. LISP is selected by default
encapsulation           = LISP
# RLOC probing configuration
#   rloc-probe-interval: interval at which periodic RLOC probes are
sent
#     (seconds). A value of 0 disables RLOC probing
#   rloc-probe-retries: RLOC probe retries before setting the locator
with
#     status down. [0..5]
#   rloc-probe-retries-interval: interval at which RLOC probes retries
are
#     sent (seconds) [1..rloc-probe-interval]
rloc-probing {
    rloc-probe-interval             = 30
    rloc-probe-retries              = 2
    rloc-probe-retries-interval     = 5
}
map-resolver        = {
        "10.0.1.1"
}
```

The configuration that follows must be different for every xTR, but it is included in the same file that the figure above:

```
##############################################
#
# xTR & MN configuration
#
# NAT Traversl configuration.
#   nat_traversal_support: check if the node is behind NAT.
nat_traversal_support  = off
# Map-Registers are sent to this Map-Server
# You can define several Map-Servers. Map-Register messages will be
sent to all
# of them.
#   address: IPv4 or IPv6 address of the map-server
#   key-type: Only 1 supported (HMAC-SHA-1-96)
#   key: password to authenticate with the map-server
#   proxy-reply [on/off]: Configure map-server to Map-Reply on behalf
of the xTR
proxy-reply    = off
map-server {
        address        = "10.0.1.1"
        key-type       = 1
        key            = lispmob
}
# IPv4 / IPv6 EID of the node.
#   eid-prefix: EID prefix (ip-prefix/mask) of the mapping
#   iid: Instance ID associated to the EID. When using VXLAN-GPE, iid
configures
#     the VNI of the mapping. [0-16777215]
# Two types of RLOCs can be defined:
#   rloc-address: Specifies directly the RLOC of the interface
#     address: It could be one of the following cases
#       - IPv4 or IPv6 address of the RLOC. Address should exist and
#        be assigned to an UP interface during startup process
otherwise
#        it is discarded.
#       - ELP name
#   rloc-iface: Specifies the interface associated with the RLOC
#     interface: interface containing the RLOCs associated to this
mapping
#     ip_version: 4 to use the IPv4 address of the interface and 6 to
use the IPv6
#       address of the interface
# Both types of RLOCs use priority and weight
#   priority [0-255]: Priority for the RLOC of the interface. Locators
#     with lower values are more preferable. This is used for both
incoming
#     policy announcements and outgoing traffic policy management.
#   weight [0-255]: When priorities are the same for multiple RLOCs,
the weight
#     indicates how to balance unicast traffic between them.
database-mapping {
    eid-prefix          = "10.244.4.1/24"
    iid                 = 0
    rloc-iface{
        interface       = ens224
        ip_version      = 4
        priority        = 1
        weight          = 100
    }
}
```

```
database-mapping {
    eid-prefix          = "192.168.8.1/24"
    iid                 = 1
    rloc-iface{
        interface       = ens224
        ip_version      = 4
        priority        = 1
        weight          = 100
    }
}
```

The other part are the files required by the driver to work. The first one if loaded by the driver to know which plugin must be used. In this case, we have chosen the plugin created to be used with OOR:

/etc/cni/net.d/10-oor.conf

```
{
  "name": "cbr0",
  "type": "oor",
  "delegate": {
    "isDefaultGateway": true
  }
}
```

The following file is loaded by the driver and contains the subnet that must be assigned to the CNI bridge and, consequently, to the containers:

/etc/oor.env

```
EID_NETWORK=10.244.0.0/16
EID_SUBNET=10.244.4.1/24
OOR_MTU=1450
OOR_IPMASQ=true
```

To sum up, it is required at least three nodes:

- A master on which the Kubernetes Control Plane will run
- A node that will join the cluster that runs in the master node and that will register an EID in the OOR MS/MR
- A different node on which OOR will run as a MS/MR

Other nodes can be added to the cluster and, consequently, others EID registered in the OOR MS/MR. In this project, we are using four servers, two of them with the role of Kubernetes Node.

16.3.2. Topology of this tests

As described in the sections above, the topology of this test consists of four servers. One of them will be dedicated to work as OOR MS/MR. The other three will have two functions in these tests: servers will play a role of OOR xTR for two EIDs, the EID of the nodes and the EID of the containers, and will also play a role within a Kubernetes environment. One of them will be the master of the Kubernetes environment, and the other two will play the role of nodes of Kubernetes. On these last servers will be where the containers of the Kubernetes cluster will be run.

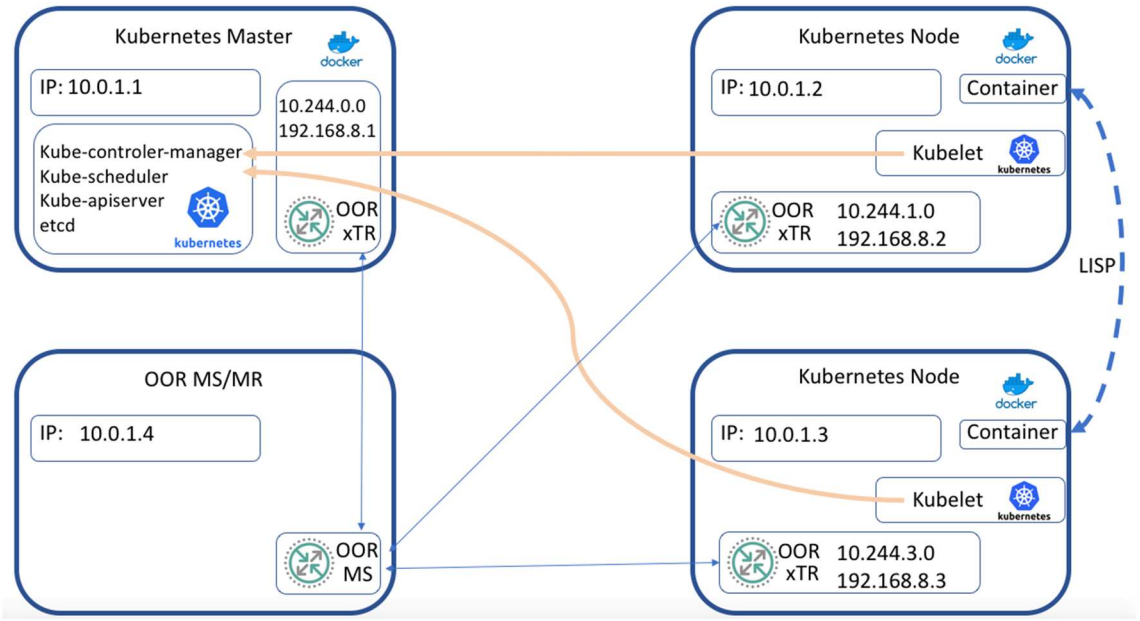

Figure 15 Test of Kubernetes and OOR - Own Compilation

16.3.3. OOR Container Native Interface Plugin Code

This is a tiny part of the project. We have decided to build a plugin based on the Bridge Container Native Plugin. We have delegated the cmdAdd and cmdDel functions to the bridge because OOR does not require any special behaviour in this part.

To simplify the documentation, and its reading, some lines of the code have been removed. The Container Native Interface developers provide a skeleton of how a CNI plugin must work. Only these two functions should be modified. The first one, cmdAdd, is the responsible of reading, the configuration of the driver, the configuration of OOR and with this information create a interface in a container and link it to a local virtual bridge.

In the following figure the code is shown:

```
func cmdAdd(args *skel.CmdArgs) error {
        net, err := loadOORNetConf(args.StdinData)
        if err != nil {
                return err
        }

        oorenv, err := loadOORSubnetEnv(net.OORFile)
        if err != nil {
                return err
        }
        if net.Delegation == nil {
                net.Delegation = make(map[string]interface{})
        } else {
                if hasKey(net.Delegation, "type") &&
!isString(net.Delegation["type"]) {
                        return fmt.Errorf("must have (string)
'type' field")
                }
        }
        // Set the attibutes required to use invoke.DelegateAdd

        net.Delegation["name"] = net.Name
        net.Delegation["type"] = "bridge"

        if !hasKey(net.Delegation, "ipMasq") {
                // we should do ipMasq
                ipmasq := !*oorenv.ipmasq
                net.Delegation["ipMasq"] = ipmasq
        }

        if !hasKey(net.Delegation, "mtu") {
                mtu := oorenv.mtu
                net.Delegation["mtu"] = mtu
        }


        if !hasKey(net.Delegation, "isGateway") {
                net.Delegation["isGateway"] = true
        }

        if net.CNIVersion != "" {
                net.Delegation["cniVersion"] = net.CNIVersion
        }

        net.Delegation["ipam"] = map[string]interface{}{
                "type":   "host-local",
                "subnet": oorenv.sn.String(),
                "routes": []types.Route{
                        types.Route{
                                Dst: *oorenv.nw,
                        },
                },
        }

        netconfMarshaled, err := json.Marshal(net.Delegation)
        if err != nil {
```

```
                return fmt.Errorf("Cannot marshal netconf: %v",
err)
        }

        // save the rendered netconf for cmdDel
        if err = saveOORAuxNetConf(args.ContainerID, net.DataDir,
netconfMarshaled); err != nil {
                return err
        }

        result, err :=
invoke.DelegateAdd(net.Delegation["type"].(string),
netconfMarshaled)
        if err != nil {
                return err
        }
        return result.Print()
}
```

The cmdDel function is responsible of removing the link, and the interface from the container:

```
func cmdDel(args *skel.CmdArgs) error {
        onc, err := loadOORNetConf(args.StdinData)
        if err != nil {
                return err
        }

        netconfMarshaled, err :=
consumeOORAuxNetConf(args.ContainerID, onc.DataDir)
        if err != nil {
                if os.IsNotExist(err) {
                        // already removed
                        return nil
                }
                return err
        }
        net := &types.NetConf{}
        if err = json.Unmarshal(netconfMarshaled, net); err != nil
{
                return fmt.Errorf("failed to parse params: %v",
err)
        }

        return invoke.DelegateDel(net.Type, netconfMarshaled)
}
```

16.3.4. How OOR Network plugin works

On one hand, to make OOR work as a Docker Driver, without using containers, with the Docker daemon standalone the next sequence must be followed:

- source /etc/oor.env
- docker daemon --bip=${EID_SUBNET} --mtu=${OOR_MTU}

- start oor

In this use case, the containers of one node cannot connect to containers running in another node. But thanks to the capabilities of OOR, this kind of connection can be created. All the containers in the node will use a bridge created by the CNI plugin, as described in the section below, as gateway. OOR must be configured in this node publishing the EID of the network to which the bridge belongs.

On the other hand, to make OOR work as a Kubernetes Driver the next sequence must be followed:

Check that the EID prefix in oor.conf is the same as the subnet assigned by Kubernetes to the node:

- Start OOR
- Start kubernetes on the node

After the following conditions, which are described in the Kubernetes Networking Model [11], are met:

- All containers can communicate with all other containers without NAT.
- All nodes can communicate with all containers (and vice-versa) without NAT.
- The IP that a container sees itself as is the same IP that others see it.

16.3.5. Additional configuration to test OOR as Kubernetes Driver

An example of each configuration required, and mentioned in this section, is shown in the previous section. They have been suppressed from this chapter to ease the reading of this report.

The binary application generated compiling the GO code above must be installed in all the servers that have a Kubernetes role. It is enough copying the binary file to all the servers.

All the nodes must have the Kubernetes installation [12] and the Docker versions that makes Kubernetes able to work. Kubernetes requires that the server where it runs does not use swap memory. So the swap has been disabled in the three servers that run Kubernetes.

In addition to this, all the servers should have installed OOR. We have deployed OOR following the guide at the wiki of the Github project [6].

The following configurations per server had been set up:

Kubernetes master:

- Kubernetes master configuration
- OOR Network Plugin configuration
- OOR Network Plugin environment configuration
- OOR working as xTR

Node 1:

- Kubernetes node configuration
- OOR Network Plugin configuration
- OOR Network Plugin environment configuration
- OOR working as xTR

Node 2:

- Kubernetes node configuration
- OOR Network Plugin configuration
- OOR Network Plugin environment configuration
- OOR working as xTR

OORNode:

- OOR working as MS/MR

In addition to this, in the latest stage of the project an ad-hoc configuration of OOR has been performed. This configuration is not shipped natively with OOR so it must be issued manually. It is a combination of the xTR mode and the Mobile Node (MN) mode of OOR. This set up has been performed in the Kubernetes master and the Kubernetes nodes to allow the communication between the containers and the nodes which is a requirement of Kubernetes for all its drivers. This configuration has to be done because the limitation of OOR to run only in one mode.

The following commands must be run to configure OOR to work in this way. They must be run in each node:

```
sudo ip addr add 192.168.8.1/32 dev lispTun0
sudo ip route add 128.0.0.0/1 dev lispTun0
sudo ip route add 0.0.0.0/1 dev lispTun0
```

16.3.6. Results of OOR as Kubernetes Driver

Doing this test was completely challenging. At the beginning of the project and during its planning stage, this stage was the only one that had risks of not achieving the objectives in the planned period.

It started with the development of the CNI Networking Plugin for Docker and Kubernetes. It was pretty much easy because the CNI developers have shipped a skeleton of how it must be and, furthermore, there are several other plugins that help to build up an idea of how to build your own Docker Networking Plugin. The tests of this part were made using the scripts also provided by the CNI developers:

```
jortiz@oor3:~/go/src/github.com/containernetworking/cni/scripts$ cat /etc/cni/net.d/10-oor.conf
{
  "name": "cbr0",
  "type": "oor",
  "delegate": {
    "isDefaultGateway": true
  }
}
jortiz@oor3:~/go/src/github.com/containernetworking/cni/scripts$ CNI_PATH=$GOPATH/src/github.com/containernetworking/plugins/bin
jortiz@oor3:~/go/src/github.com/containernetworking/cni/scripts$ sudo CNI_PATH=$CNI_PATH ./docker-run.sh --rm busybox:latest ip route
default via 10.244.3.1 dev eth0
10.244.0.0/16 via 10.244.3.1 dev eth0
10.244.3.0/24 dev eth0 scope link  src 10.244.3.11
jortiz@oor3:~/go/src/github.com/containernetworking/cni/scripts$
jortiz@oor3:~/go/src/github.com/containernetworking/cni/scripts$
```

Figure 16 CNI Plugin Test

Own compilation

When this part was finished and tested, the following part has to start with the Kubernetes installation and configuration which has been detailed in previous sections above, resulting in a fully working Kubernetes environment. To show the behaviour of this cluster the following images are provided.

The following image shows the successful installation of Kubernetes with the networking driver. If the Networking driver is not correctly set up, this initialization will not finish correctly:

```
kubeadm init --pod-network-cidr=10.244.0.0/16 --apiserver-advertise-address=84.88.81.77 --skip-preflight-checks
Flag --skip-preflight-checks has been deprecated, it is now equivalent to --ignore-preflight-errors=all
[init] Using Kubernetes version: v1.9.0
[init] Using Authorization modes: [Node RBAC]
[preflight] Running pre-flight checks.

[addons] Applied essential addon: kube-proxy

Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Figure 17 kubernetes initialization

Own compilation

The following image shows an usual kubernetes namespace:

```
root@oor1:~# kubectl get pod -n default -o wide
NAME                                READY     STATUS    RESTARTS   AGE       IP           NODE
nginx-deployment-6c54bd5869-647p8   1/1       Running   0          1d        10.244.3.6   oor3
nginx-deployment-6c54bd5869-9kb6d   1/1       Running   0          7s        10.244.4.6   oor2
nginx-deployment-6c54bd5869-fx9vh   1/1       Running   0          1d        10.244.3.5   oor3
```

It shows that two pods are running in the oor3 node and another node that runs in oor2. The following figure shows the connectivity from the pod of oor2 to the pod of oor3 using LISP and OOR:

```
root@oor1:~# kubectl get pod -n default -o wide
NAME                                READY     STATUS    RESTARTS   AGE       IP           NODE
nginx-deployment-6c54bd5869-647p8   1/1       Running   0          1d        10.244.3.6   oor3
nginx-deployment-6c54bd5869-9kb6d   1/1       Running   0          18m       10.244.4.6   oor2
nginx-deployment-6c54bd5869-fx9vh   1/1       Running   0          1d        10.244.3.5   oor3
root@oor1:~# kubectl exec -it nginx-deployment-6c54bd5869-9kb6d bash -n default
root@nginx-deployment-6c54bd5869-9kb6d:/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0@if78: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue state UP
    link/ether 0a:58:0a:f4:04:06 brd ff:ff:ff:ff:ff:ff
    inet 10.244.4.6/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::90fe:11ff:fe49:5dbf/64 scope link
       valid_lft forever preferred_lft forever
root@nginx-deployment-6c54bd5869-9kb6d:/# ping 10.244.3.6
PING 10.244.3.6 (10.244.3.6): 48 data bytes
56 bytes from 10.244.3.6: icmp_seq=0 ttl=64 time=0.128 ms
56 bytes from 10.244.3.6: icmp_seq=1 ttl=64 time=0.082 ms
56 bytes from 10.244.3.6: icmp_seq=2 ttl=64 time=0.087 ms
56 bytes from 10.244.3.6: icmp_seq=3 ttl=64 time=0.079 ms
56 bytes from 10.244.3.6: icmp_seq=4 ttl=64 time=0.084 ms
^C--- 10.244.3.6 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.079/0.092/0.128/0.000 ms
```

Figure 18 Kubernetes OOR Connection

Own compilation

In addition, the connection between the containers and the nodes can be established through OOR:

```
root@nginx-deployment-6c54bd5869-647p8:/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
      valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
      valid_lft forever preferred_lft forever
3: eth0@if158: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue state UP
   link/ether 8a:40:43:e5:fa:98 brd ff:ff:ff:ff:ff:ff
   inet 10.244.3.6/24 scope global eth0
      valid_lft forever preferred_lft forever
   inet6 fe80::8840:43ff:fee5:fa98/64 scope link
      valid_lft forever preferred_lft forever
root@nginx-deployment-6c54bd5869-647p8:/# ping 192.168.8.2
PING 192.168.8.2 (192.168.8.2): 48 data bytes
56 bytes from 192.168.8.2: icmp_seq=0 ttl=62 time=2.614 ms
56 bytes from 192.168.8.2: icmp_seq=1 ttl=62 time=1.128 ms
56 bytes from 192.168.8.2: icmp_seq=2 ttl=62 time=1.153 ms
56 bytes from 192.168.8.2: icmp_seq=3 ttl=62 time=1.190 ms
56 bytes from 192.168.8.2: icmp_seq=4 ttl=62 time=1.165 ms
```

The following figures show that the connection between them is made using LISP.

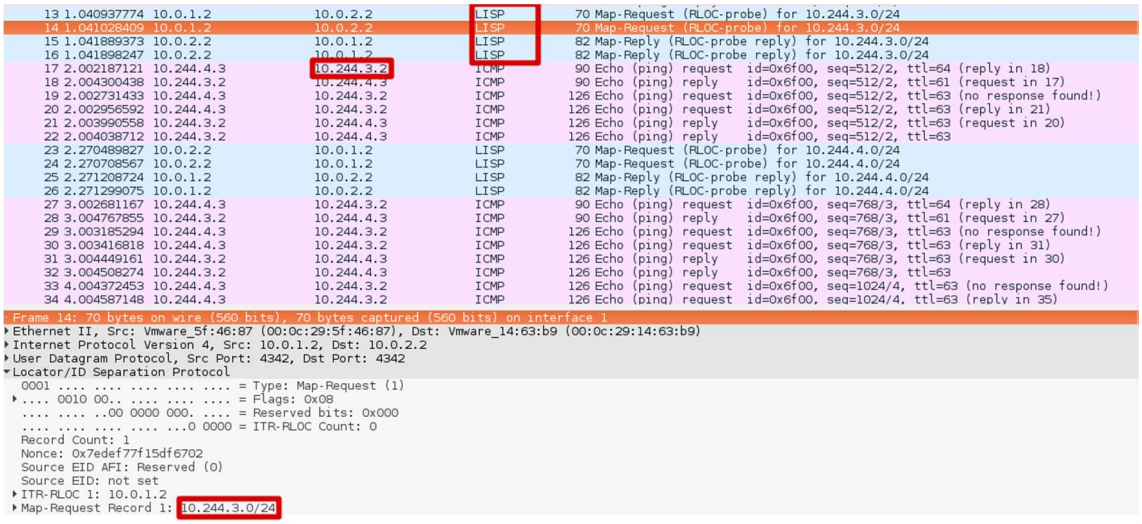

Figure 19 Kubernetes Wireshark Capture (I)

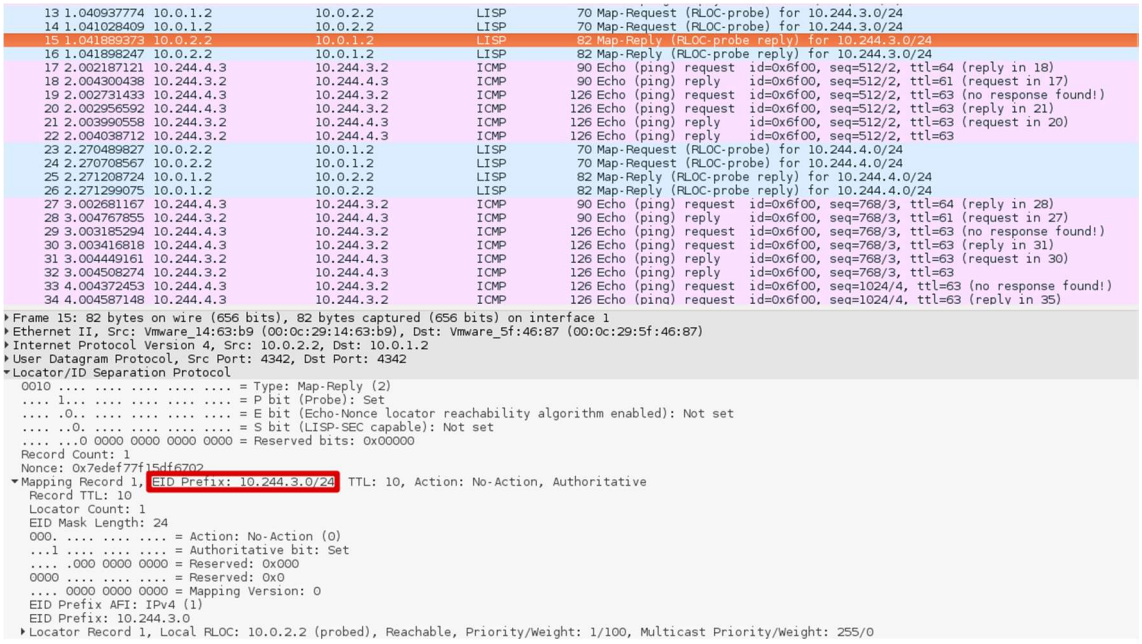

Figure 20 Kubernetes Wireshark Capture (II)

The Kubernetes model for their networking drivers talk about the connection from containers to nodes and vice-versa. As is shown above, the connection from the containers to the nodes is working, and the connection from the nodes to the containers is also working.

This can be shown in the following figure:

```
root@oor1:~# kubectl get pod -n default -o wide
NAME                                READY     STATUS    RESTARTS   AGE       IP            NODE
nginx-deployment-6c54bd5869-ctrm4   1/1       Running   0          14h       10.244.3.12   oor3
nginx-deployment-6c54bd5869-sglk4   1/1       Running   0          14h       10.244.3.13   oor3
nginx-deployment-6c54bd5869-tlrbk   1/1       Running   0          14h       10.244.3.14   oor3
root@oor1:~# kubectl exec -it nginx-deployment-6c54bd5869-tlrbk ip a -n default
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0@if167: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1450 qdisc noqueue state UP
    link/ether f6:2f:5a:9a:90:4f brd ff:ff:ff:ff:ff:ff
    inet 10.244.3.14/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::f42f:5aff:fe9a:904f/64 scope link
       valid_lft forever preferred_lft forever
root@oor2:~# ip route get 10.244.3.14
10.244.3.14 dev lispTun0  src 192.168.8.2
    cache
root@oor2:~# ping 10.244.3.14
PING 10.244.3.14 (10.244.3.14) 56(84) bytes of data.
64 bytes from 10.244.3.14: icmp_seq=1 ttl=62 time=0.958 ms
64 bytes from 10.244.3.14: icmp_seq=2 ttl=62 time=1.36 ms
64 bytes from 10.244.3.14: icmp_seq=3 ttl=62 time=0.890 ms
^C
--- 10.244.3.14 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 0.890/1.070/1.364/0.212 ms
```

Figure 21 Kubernetes Node To Pod

So all the features required by Kubernetes to act as a Network Plugin are working.

17. Conclusions

On one hand, the first conclusion we can extract after reviewing the results of the tests is that a new paradigm of development can be established for OOR easily. It is just required that someone follow the steps described in the documentation on how to build a complete CI/CD lifecycle which is available in the Internet. Furthermore, researchers and other developers that help in the development of OOR have now more tools to automate the tests and deployments to production environments.

On the other hand, the conclusions we can extract after reviewing the results of the OOR and Kubernetes tests is that OOR is not yet ready to work as a Kubernetes Networking Plugin. It is required to achieve the full connection from the containers to the nodes and vice-versa, without any manual configuration.

To be able to work like OVS, developers and researches of OOR should develop a feature that allow the connection from the nodes to the containers.

Regarding to the project planning as it is described in the previous report, almost all the milestones have been achieved in time. The stage of OOR Containerization had lasted longer than the expected duration. The reason for that are new requirements of the OOR Container requested by the researchers. To avoid delays of the project, the amount of time that was forecasted for possible deviations in the "OOR as Kubernetes Networking Driver" stage has been used for the mentioned stage.

18. Future Work

From the point of the OOR in containers, there is not such amount of work to do. Some improvements can be done in order to increase the performance and to reduce the weight of the image. Furthermore, other requirements might be requested by the community or the researchers.

From the perspective of OOR as Kubernetes Networking Plugin, there are some improvements required to allow OOR work in this way. One of the improvements that will help is to implement a DB where dynamically set the bootstrap configuration. This improvement should have the same features as OVS-DB. In addition to this, this database should have the ability to send requests to etcd or consul (in which Kubernetes rely on to work) to ask for subnet leases. The other feature that OOR must acquire is the ability to work like the configuration that has been made during this project manually to make OOR work as a Kubernetes cluster. This means that OOR must be able to work in the mixed mode xTR-MN described above

These two features will make OOR as competitive as OVS in Kubernetes networking.

Acknowledgements

To my family whose faithfully belief that I am able to finish my degree has encouraged me to also finish this project.
To Berta whose endless love has helped me to overcome any technical problem during the development of this project.
To Albert Lopez Brescó whose amazing knowledge of OOR and wisdom in Networking has let me be quiet and confident while developing this project.
To both Albert and Jordi (the Director of the project and the co-director) whose patience with me has been infinite.

References

[1] K. Authors, https://kubernetes.io/docs/admin/ovs-networking/, URL: Kubernetes (Last visited January 2018).

[2] LISP, https://www.lisp4.net, URL: LISP, visited September 26th 2017.

[3] Docker, Docker for the Virtualization Admin, San Francisco: Docker eBook, 2016.

[4] SdxCentral, https://www.sdxcentral.com/sdn/definitions/inside-sdn-architecture/, URL: sdxcentral, visited September 26th 2017.

[5] D. Meyer, cator/ID Separation Protocol Locator/ID Separation Protocol (LISP) Tutorial (LISP) Tutorial, Vancouver: IETF, 2007.

[6] A. L. Brescó, https://github.com/OpenOverlayRouter/oor/wiki, Barcelona: URL, 2017.

[7] J. P. A. L.-B. A. C. e. a. Alberto Rodriguez-Natal, Programmable Overlays via OpenOverlay Router, Barcelona: UPC, 2017.

[8] OVS, https://en.wikipedia.org/wiki/Open_vSwitch, URL: OVS, visited on September 26th 2017.

[9] C. Tozzi, Why Docker Containers are Good for the Environment, URL: Container Journal, Last visited October 22nd, 2017.

[10] Docker, docs.docker.com, web: Docker.

[11] T. K. Authors, Kubernetes model, web: https://kubernetes.io/docs/concepts/cluster-administration/networking/#kubernetes-model, 2018.

[12] K. Authors, https://kubernetes.io/docs/getting-started-guides/scratch/, URL: Internet (Last visited January 2018).